5-1-2013

Scalable Synchronization with Mindicators

Logan McNamara
Lehigh University

Yujie Liu
Lehigh University

Scalable Synchronization with Mindicators

(Regular Submission)

Logan McNamara
Lehigh University
lrm213@cse.lehigh.edu

Yujie Liu
Lehigh University
yul510@cse.lehigh.edu

Michael Spear
Lehigh University
spear@cse.lehigh.edu

Abstract

The Mindicator is a shared object that stores one value for each thread in a system, and can return the minimum of all thread's values in constant time. In this paper, we explore applications of the Mindicator in synchronization algorithms. We introduce three new algorithms, designed for scalable Read-Copy-Update (RCU), fair Readers-Writer locking, and Group Mutual Exclusion. Experimental evaluation shows these algorithms to perform well while avoiding contention.

Keywords: Readers-Writer Locking, Group Mutual Exclusion, Read-Copy-Update, Linearizability, Fairness

1 Introduction

The goal of a synchronization object is to ensure a globally consistent order among the invocations and responses of its methods, as made by multiple threads. This order induces a happens-before relationship among the threads, through which incompatible operations on some other data structure can be sequenced in a manner suitable for ensuring program correctness.

Although these synchronization objects are inherently concurrent, they need not be scalable. As a simple example, consider a mutual exclusion lock: since it protects operations that run sequentially, there is little benefit in coordinating concurrent lock acquisition attempts: after an attempt completes and an order is determined, most of the threads will not be able to progress. While low latency `Acquire()` and `Release()` operations are important, the cleverness of determining an order through some scalable concurrent algorithm offers little benefit to overall program execution time.

As a result, most research into scalable lock implementations focuses on decreasing contention over shared memory locations. The most popular approach is embodied by variations on the MCS lock [26]: threads serialize via modifications to a single location when they add themselves to the tail of a queue, but subsequent interaction among threads to coordinate entry to critical sections is achieved through decentralized communication across $O(\#threads)$ unique memory locations. In addition to low contention, queue-based locks can be naturally extended to support more complex forms of synchronization, such as Readers-Writer locks and group mutual exclusion. Furthermore, these implementations tend to be fair, since the queue is a first-in first-out data structure. When the more scalable first-in, first-enabled guarantee is desired [16], this too can be achieved via a simple modification to the logic of the queue [21].

Unfortunately, queue-based synchronization has several bottlenecks. Consider a Readers-Writer lock implementation [4]. Even in the absence of writer threads, all readers must contend over the same location in order to enqueue themselves. Then, when the readers have reached the end of their critical sections, a cascading series of updates to queue nodes is required before a writer can commence.

In this paper, our focus is on synchronization paradigms in which several threads can make progress in critical sections simultaneously, specifically read-copy-update synchronization, Readers-Writer locks, and group mutual exclusion. In these settings, it is desirable for threads to avoid updating the same location at the beginning of their critical sections, since they can otherwise execute in parallel without conflicting memory accesses. It is also useful to have a broadcast mechanism for notifying a group of threads when they may begin execution.

Our new algorithms exploit the Mindicator data structure [23]. Briefly, the Mindicator allows a set of threads to each maintain a single value within it, such that changing a thread's value takes $O(\#threads)$ time, and determining the minimum value across all threads' values can be done in a single instruction. As a highly scalable data structure with low memory contention, the Mindicator allows us to craft decentralized approaches to our target synchronization problems.

In a break with prior work, we relax some progress guarantees of the synchronization objects in order to achieve better scalability. As an example, consider a common wait-free solution to group mutual exclusion [16]: to achieve a wait-free ordering of an operation, a mechanism similar to Lamport's bakery lock [20] is employed, where every instruction must access $\#threads$ locations, potentially incurring $\#threads$ cache misses. By using a lock-free Mindicator and a spinlock, we can reduce this cost to $O(log(\#threads))$, with several common cases requiring only a constant number of memory accesses.

The remainder of this paper is organized as follows. In Section 2 we give background on the Mindicator object. Sections 3- 5 present and evaluate new Mindicator-based algorithms for read-copy-update, Readers-Writer locks, and group mutual exclusion, respectively. Section 6 concludes.

2 Background: The Mindicator

A Mindicator [23] is a shared object that can compute the minimum value over a set of per-thread values in a lock-free manner. A Mindicator object exposes three methods: ARRIVE(v) is used to set

a thread's value to v. DEPART() is used to set a thread's value to $\top$ (the maximum possible value). Successive ARRIVE() operations without an intervening DEPART() are not permitted. QUERY() returns the minimum value across all thread's values.

Similar to SNZI [10] and f-arrays [15], a Mindicator object is structured as a tree, with the root of the tree at all times maintaining the minimum. Typically, every thread in a system is assigned a dedicated leaf in the tree, into which it can store a single value, or $\top$. In an ARRIVE operation, the thread will iteratively pass its value upward to each ancestor a_i of its leaf if its value is the minimum over all descendants of a_i. In a DEPART operation, the thread will update each of its ancestors to ensure that the ancestor stores the smallest value among all its descendants. A careful lock-free protocol summarizes the minimum of intermediate trees, thereby ensuring that most operations terminate their propagation without modifying the root node. This enables good scalability by limiting data sharing and cache coherence traffic.

3 RCU Synchronization

We begin analyzing the applicability of Mindicators to read-copy-update (RCU) synchronization. Originally proposed as a synchronization mechanism for certain operating system functionality [12, 24], RCU has grown into a more generally applicable, though still not fully generalizable, approach to scalable synchronization [7].

3.1 RCU Background

RCU operations are partitioned into two classes: wait-free read-only operations, and blocking writer operations. A read-only operation proceeds by setting some flag, performing its operation, and then clearing its flag; this has negligible overhead. Writers are typically serialized via a programmer-supplied lock. Both read-only and writer operations must follow application-specific conventions. For example, a doubly-linked list may require writers to only use certain swap operations to change values, and may forbid readers from using back links.

Even under such constraints, a writer must often split its operation into two portions. The "first half" typically makes certain data unreachable by future operations, and the "second half" requires a guarantee that no in-flight read operations still possess a pointer to the unreachable data. A sufficient condition is to delay the second half until the writer is certain that every incomplete read operation started after the writer completed the first half. To enforce this ordering, a writer may call a WRITERSYNC operation while holding the lock, which entails reading the flags of every reader and blocking until each thread's flag changes in a manner that indicates progress outside of a read-only critical section. In the most simple case, this is achieved by having read-only critical sections increment their per-thread counters to odd when they begin an operation, and again to even when they complete. A writer synchronizes by waiting for each flag to either become even or change.[1]

While RCU favors workloads with an extremely high incidence of read-only operations, it has been shown to work well even when writers are more frequent [7]. Indeed, when all threads are writers, RCU performance should be equivalent to that of the application-specific mutual exclusion lock.

3.2 The Mindicator-RCU Algorithm

In a typical RCU implementation, the per-thread counters are not synchronized, and thus $O(\#threads)$ locations must be checked during a WRITERSYNC operation, since it must read every thread's counter. Furthermore, the application-specific lock is completely decoupled from the RCU mechanism. For *kernel-space* RCU, these design decisions enable a single counter for each kernel thread to support multiple independent RCU writers, each synchronized with a different set of locks, while also ensuring wait-freedom for readers. Inasmuch as RCU reader code may exist within interrupt handlers, wait-freedom is essential. However, for userspace RCU these properties are not necessarily advantageous.

We provide a userspace RCU interface consisting of seven operations, as described in Figure 1. Our interface extends the kernel interface by two operations, representing the integration of a writer lock into the RCU API. Note that this interface does not

[1] A similar mechanism is used for memory reclamation in transactional memory implementations [11, 14].

alter any fundamental aspects of RCU synchronization. In particular, critical sections (writer and read-only) must follow application-specific conventions to preserve correctness.

Our implementation employs two data structures. The first is an integer $counter$ that can be read, set, and incremented atomically using load, store, and compare-and-swap (CAS) operations, respectively. The second is a $mindicator$. For this discussion, we assume that $mindicator$ is preconfigured to a size suitable for the number of threads in the application. Given such a Mindicator, the REGISTER and UNREGISTER functions respectively reserve or release a reservation on a unique leaf of $mindicator$ for the calling thread.

The implementation of the remaining five functions appears in Algorithm 1. The $counter$ serves as a lock, with odd values indicating that the lock is held. This suffices to provide mutual exclusion. Readers, for their part, do not block when the lock is held. They simply register in $mindicator$ the value they observed. Since $counter$ increases monotonically, the values saved by successive calls to READERLOCK will be monotonically increasing.

Note that with this implementation, the progress guarantees for readers reduce to the progress guarantees afforded by the Mindicator (typically lock-freedom). Furthermore, note that the Mindicator provides the exact same specification as a traditional RCU implementation (with per-thread counters). In particular, the same correctness argument applies to the sole corner-case in the implementation. Suppose that a reader delays between its first and second instructions in READERLOCK, and during this delay all other readers complete and a writer executes WRITERSYNC. In such a situation, the call to WRITERSYNC could return immediately, even though an instant later, the reader could resume and complete, such that a subsequent QUERY would return a value smaller than $counter$. In this case, since WRITERSYNC completion precedes the completion of the reader's ARRIVE, the reader's critical section is guaranteed to only see state that was installed prior to the WRITERSYNC call. Thus while such "flicker" will delay subsequent writers, it will not compromise the correctness of the application: the reader is ordered after the writer, even if the QUERY operations suggest otherwise.

Algorithm 1: RCU Locking with Mindicators

```
   shared data
     counter    :  Integer
     mindicator :  Mindicator

 1 procedure WRITERLOCK()
 2     while true do
 3         c ← counter
 4         if c mod 2 = 0 then
 5             if CAS(&counter, c, c + 1) then
 6                 break

 7 procedure WRITERUNLOCK()
 8     counter ← counter + 1

 9 procedure WRITERSYNC()
       // increment counter while holding the lock
10     c ← counter
11     counter ← c + 2
       // wait for existing readers to depart
12     while mindicator.QUERY() < c + 2 do wait

13 procedure READERLOCK()
14     c ← counter
15     mindicator.ARRIVE(c)

16 procedure READERUNLOCK()
17     mindicator.DEPART()
```

3.3 Performance

To evaluate the effectiveness of the Mindicator in an RCU implementation, we compare it to the baseline userspace RCU implementation, version 0.7.6. The userspace RCU package includes a microbenchmark parameterized by the number of read-only threads, number of writer threads, duration of each read-only critical section, duration of each writer critical section, and the interval between writer critical sections. The microbenchmark is somewhat artificial, in that every interval is achieved by executing a tight loop of no-op instructions, without any memory accesses or other nondeterministic operations. Nonetheless, it is a useful stress test, since the overheads of the RCU mechanism will be exaggerated.

We configured readers to perform 256 no-op instructions and writers to perform 512 no-ops. Figure 2 presents the results. In the leftmost chart, there are no writer operations. The middle chart presents performance when writers incur a delay of 32K cycles between critical sections, and the rightmost chart

Function	Purpose
REGISTER	Registers a thread with an RCU lock, so that its state is visible to writers.
UNREGISTER	Deregister a thread, so that writers of the associated RCU lock will no longer wait for it in WRITERSYNC.
WRITERLOCK	Acquire this RCU lock's mutex.
WRITERSYNC	Delay the writer thread until any active readers have completed.
WRITERUNLOCK	Release this RCU lock's mutex.
READERLOCK	Make this thread visible to any concurrent writer. Called before a read-only critical section.
READERUNLOCK	Indicate that this thread is no longer reading data protected by the RCU lock. Called at the end of a read-only critical section.

Figure 1: Extended RCU interface

presents an experiment with no delay between writer critical sections. The total number of completed read and write critical sections are charted for both the original userspace RCU implementation (orig) and our Mindicator-based RCU algorithm (mind). The Mindicator was configured with 64 leaves.

In the (orig) experiments, read-only critical sections set a per-thread variable before beginning, and then set it again upon completion. In contrast, the Mindicator requires $O(log(n))$ compare-and-swap (CAS) operations, where n is the maximum number of threads supported (in this case, 64). Of course, in the common case, many fewer CASes will be needed, since most Mindicator operations will not traverse upward all the way to the root node. The comparable throughput of the Mindicator-based RCU implementation, even for the read-only workload, suggests that on modern architectures, adding a handful of uncontended CAS operations should not be a significant obstacle to performance.

Similarly, writer throughput is equivalent in both implementations. When writers incur a delay between executing, both implementations achieve a flat rate of around 34K write operations over the course of the 5-second experiment. This suggests that the implementation of the mutex does not affect overall throughput; the delay incurred in WRITERSYNC should dominate the cost of write acquisition. Even when writes become more frequent, we see no difference between in writer throughput. While impressive, we caution that this result should not be generalized too far: When the microbenchmark is run with one writer thread, the writer will never fail to acquire the write lock on its first try.

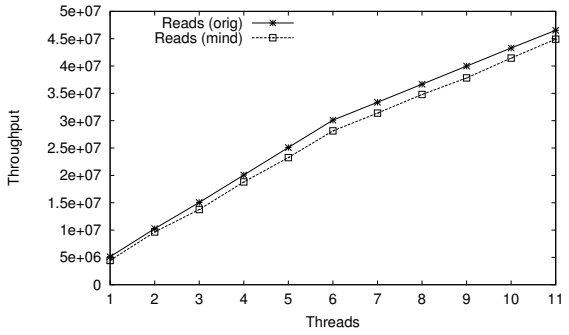
3.4 Discussion

At first glance, it does not appear that integrating the mutex into the RCU interface, and then employing a Mindicator to streamline the implementation of WRITERSYNC, offers a performance benefit: an asymptotic improvement to writer overhead does not matter, since writers are exceedingly rare. However, there is no significant performance loss, despite adding profound new capabilities.

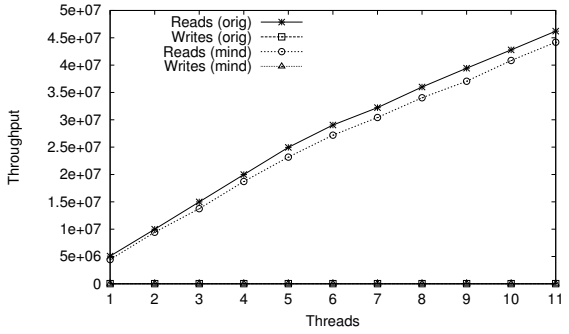
To highlight the benefit, we observe that our modification makes RCU an overlay atop of a counter lock (i.e., a sequence lock [19]). This, in turn, means that the same lock word can be repurposed into multiple modes, depending on application state. For example, by reserving two bits in the lock word to indicate state, it would be possible to use the same lock to mediate critical sections running as transactions using the NOrec algorithm [6], sequence locks, a traditional odd/even mutex, or RCU. Considering the ever-growing role of auto-tuning in parallel systems, the ability to repurpose a lock for different synchronization modes is an appealing possibility, which we believe justifies the cost.

4 Readers-Writer Locking

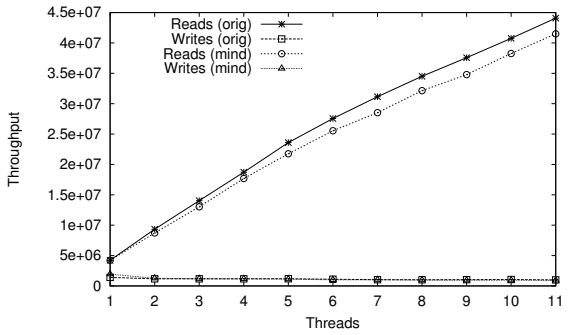
Readers-Writer locks [4] provide a more general form of synchronization than RCU. A Readers-Writer lock allows either a single write operation, or a set of read-only operations, to be in a critical sec-



(a) Reader Throughput (no writes)



(b) Reader and Writer Throughput (sporadic writes)



(c) Reader and Writer Throughput (continuous writes)

Figure 2: RCU reader throughput for different writer frequencies

tion at any one time. Unlike mutual exclusion locks, this behavior necessitates a more scalable implementation of the lock itself: if multiple read-only critical sections attempt to acquire the lock simultaneously, they should not contend over the lock at either boundary of their critical section.

Readers-Writer lock implementations differ in terms of the guarantees that they provide. The most fair implementations provide a first-in-first-out guarantee. That is, if thread A's attempt to acquire the lock linearizes before thread B's attempt, and A and B are not both attempting to acquire the lock as readers, then A's critical section must execute before B's critical section. Of course, if both threads are read-only operations, their critical sections should run concurrently.

The best known fair readers-writer lock implementations enqueue all lock acquire requests in a queue [18, 25]. When a request reaches the head of the queue, the associated thread may execute its critical section. In addition, a reader may enter its critical section if the node before it in the queue is a reader that is allowed to enter its critical section. When a critical section completes, the associated thread removes its entry from the queue. Note that there is some complexity in how queue nodes are released, since readers do not necessarily complete their critical sections in the same order as their lock acquisition requests were enqueued.

Recently the OLL family of locks was introduced as a scalable readers-writer lock implementation [21]. The key innovation in these locks is that successive read requests do not lead to multiple enqueues. Instead, queue nodes contain a scalable counter-like object [10]. In a manner similar to reference counting, attempts to acquire the lock in read-only mode will increment the counter in the tail node (if it corresponds to a reader) rather than enqueueing a new node. Since the counter itself does not cause contention, and readers are less likely to modify the tail pointer of the queue, the result is much less contention, and hence higher performance.

4.1 Algorithm Sketch

The key innovation that our Mindicator Readers-Writer lock introduces is a decentralized queue. Observe that in the OLL locks, enqueue operations are totally ordered, and the mechanism for indicating when a critical section may begin is simply that the corresponding node is at the head of the queue.

Based on this observation, our implementation uses two objects: a shared counter and a Mindica-

tor. The counter is initially even. To enqueue a write request, a thread atomically reads the counter (as c), puts the value $c + 1$ in the Mindicator, and updates the counter value to $c + 2$. Note that in this idealized presentation, the counter never takes an odd value, so any odd value in the Mindicator must correspond to exactly one writer operation. To enqueue a read request, a thread atomically reads the counter (as c) and sets its value in the Mindicator to c. Threads then query the Mindicator repeatedly until they observe their value as the minimum. At that point, the critical section may run, after which time the thread removes its value from the Mindicator, replacing it with $\top$.

Note that with this algorithm, readers *never* modify the counter, and space overhead a fixed function of the number of threads. Threads have no knowledge of their place in the queue (e.g., a writer cannot tell if the preceding operation is a reader or writer); they only know which set of operations is at the head of the queue, and when they are permitted to run.

4.2 Algorithm Details

Algorithm 2 presents a practical implementation of the idealized readers-writer lock, using only load, store, and compare-and-swap (CAS) operations. Instead of atomically modifying both the counter and Mindicator, we use the least significant bit of the counter as a lock: when the counter is odd, a writer is in the process of enqueueing itself, blocking all other writers from enqueueing themselves.

To prevent writers from blocking readers, or readers from blocking writers, we employ a slightly more complex protocol for readers. To handle the case when a reader attempts to acquire the lock while a writer is in the midst of an acquire operation (e.g., the counter is odd), we instruct the reader to use the next even value, rather than the current value (Line 15). Note that this is the only condition that can lead to Lines 18–20 executing. These lines ensure that if the writer delays for long enough for the reader to complete its arrival, the reader does not query the Mindicator until the writer finishes. This is necessary. Otherwise, the reader could progress past Line 27 before the writer completed Line 6. In this case, the reader and writer could enter their critical sections simultaneously, violating mutual exclusion.

Algorithm 2: Readers-Writer Locking with Mindicators

```
   shared data
     counter    :  Integer
     mindicator :  Mindicator
 1 procedure WriterLock()
 2     while true do
 3         c ← counter
 4         if c mod 2 = 0 then
 5             if CAS(&counter, c, c + 1) then
 6                 mindicator.Arrive(c + 1)
 7                 counter ← c + 2
 8                 break
 9     while mindicator.Query() ≠ c + 1 do wait
10 procedure WriterUnlock()
11     mindicator.Depart()
12 procedure ReaderLock()
13     while true do
14         c ← counter
15         d ← (c mod 2 = 0) ? c : c + 1
16         mindicator.Arrive(d)
17         c ← counter
18         if c < d then
19             while counter < d do wait
20             break
21         else if c = d then
22             break
23         else if c = d + 1 then
24             break
25         else if c > d + 1 then
26             mindicator.Depart()
27     while mindicator.Query() ≠ d do wait
28 procedure ReaderUnlock()
29     mindicator.Depart()
```

Since a writer can request the lock while a reader is at any step in the ReaderLock operation, we must also ensure that if a reader delays on Line 16, that it does not lower the value of the Mindicator after some other critical section has begun. The default, on Line 21, is that no writer who orders after the reader has attempted to acquire the lock. If, however, a single writer W ordered after the reader has attempted to acquire the lock (indicated by the counter between Lines 5 and 7) but W has not called Query, the reader may continue (as indicated by Lines 23–24). Otherwise (Lines 25–26) the reader

cannot be sure that a concurrent writer has not yet completed Line 9. In this case the reader removes its value from the Mindicator and retries its operation.

4.3 Blocking vs. Spinning

While this algorithm naturally lends itself to spin locks, it can be extended to support blocking (yielding the CPU) with minimal effort and without introducing a scalability bottleneck. Since every write operation or set of read operations has a unique integral value, we can create a semaphore for each value, and use the value as a hash key for locating the semaphore in $O(1)$ time. When a writer (or last reader) departs, it determines if there is a pending semaphore and performs an $up()$ on it. When a QUERY does not allow a thread to progress, it creates or locates a semaphore, calls QUERY again, and if it still cannot progress, performs a $down()$ on the semaphore. Upon waking, a thread wakes fellow readers (if any) by performing an $up()$ on the semaphore. The value of a QUERY can be used at any time to determine which semaphores are unreachable and can be garbage collected.

4.4 Performance

We evaluate our Readers-Writer lock through a stress-test microbenchmark. We use the same machine and configuration as in Section 3.3. We also compare a larger range of algorithms.

Curves labeled "Min" use our Mindicator-based locks. These are configured with 64 leaves. The "CAS" curves use a single word to encode an unfair readers-writer lock with minimal latency. In this implementation, the least significant bit indicates a writer, and the remaining bits a count of the readers. The lock is biased toward writers. The "pthread_rwlock" and "pthread_mutex" curves show the performance of pthread readers-writer locks, and pthread mutex locks, respectively. Finally, "f-array" depicts the performance of our algorithm using a wait-free f-array in place of the lock-free Mindicator.

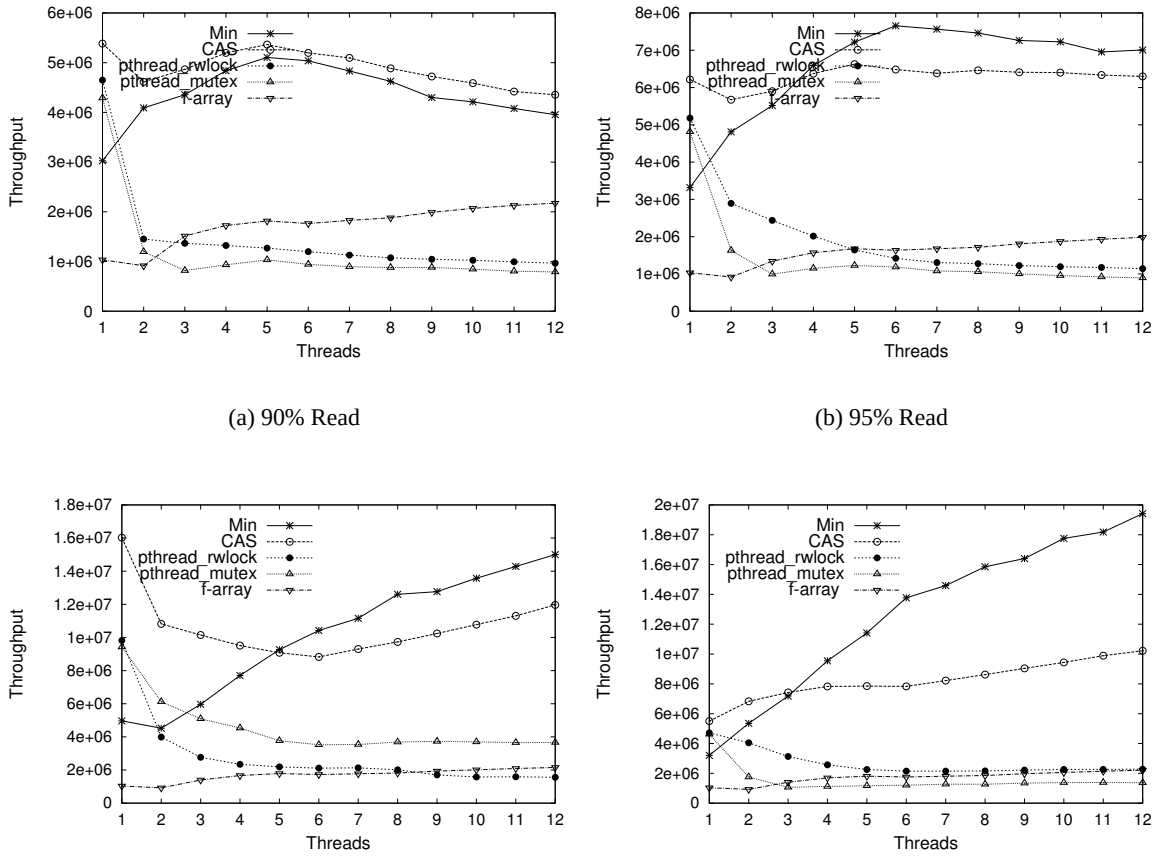
We present stress-test results, in which threads repeatedly acquire and release the lock with no intervening operations. Unlike the RCU tests, write requests are distributed among the threads, so that there can be real contention and serialization among writers. The tests are parameterized by the likelihood of requesting the lock in read-only mode. Figure 3 presents performance for read-only ratios of 100%, 99%, 95%, and 90%. Every data point is the average of 5 trials.

At low thread counts, the CAS lock achieves the best performance. This is no surprise, since it entails the fewest instructions. However, as the read ratio increases, the scalability afforded by the Mindicator decreases the thread count at which the Mindicator lock performs better. Among the most significant points is that simultaneously releasing a read lock does not cause threads to contend over a memory location, particularly since they do not access the counter at all. In contrast, the CAS implementation requires contended accesses to the counter for every acquire and release. Thus even in the pure read-only case, the CAS lock does not scale well.

Another important result that follows from these experiments is that the Mindicator lock performs well both because of the high-level algorithm, and the internal details of the Mindicator. In particular, when we used a more strict Mindicator implementation, in which a depart cannot terminate as early, scalability dropped by more than half. Similarly, using an f-array in place of the Mindicator, but otherwise keeping the algorithm the same, leads to significantly worse performance.

4.5 Discussion

Our algorithm introduces asymmetry in its progress guarantees. Lock release operations offer the same guarantees as the underlying Mindicator, and read lock acquisition composes a lock-free approach with the Mindicator ARRIVE operation's guarantees. For the Mindicators considered in this paper, the result is lock-free: a read lock acquisition never blocks, never causes another operation to block, but may starve due to concurrent write lock acquisitions. Write lock acquisitions are blocking with respect to each other, but do not impede the lock-freedom of requesting reader locks. Furthermore, the act of requesting the lock is orthogonal to the act of being granted the lock. Once a lock request is enqueued in the Mindicator, subsequent lock requests will not interfere with the completion of critical sections, or the hand-off of the lock



(a) 90% Read

(b) 95% Read

(c) 99% Read

(d) 100% Read

Figure 3: RWLock Lock Performance

from one thread to another.

Our lock is also relatively contention-free. Serialization on a lock word (in our case, the counter) for writers is, of course, inevitable. However, readers do not modify this lock word, and their arrivals in the Mindicator are as scalable as the Mindicator itself. Furthermore, lock release is simple, since there is no complexity due to readers completing their critical sections in an arbitrary order, and the algorithm itself does not entail any memory reclamation. This latter point is significant, since memory reclamation bugs have recently been found in the Kreiger lock [8].

5 Group Mutual Exclusion

Readers-Writer locks can be thought of as a specific class within the larger group mutual exclusion problem [3, 13, 16, 17]. In group mutual exclusion, each thread is dynamically assigned to a session. Critical sections of threads belonging to the same session may run concurrently, but critical sections of threads belonging to different sessions may not. By this formulation, a traditional mutual exclusion lock is one in which every thread belongs to its own unique session, and a readers-writer lock is one in which all read-only operations are performed by threads of the same session and all other operations (write operations) are performed by threads belonging to unique sessions.

It is trivial to extend our readers-writer algorithm to a *completely fair* group mutual exclusion algorithm. However, such an extension misses a key opportunity. Suppose that threads in session A are executing their critical sections, and threads in sessions

B and C are waiting their turn. If a new operation of session B wishes to run, it may not be desirable to create a second session of type B ordered after session C since the operation can run concurrently with all other threads of session B; instead the new operation should be allowed to join the existing B session. Intuitively, this captures the notion that allowing the operation to bypass those in C will not delay the execution of operations in C, but will improve overall throughput.

While group mutual exclusion has not seen much practical use, it is likely to become significant in future systems. For example, scheduling algorithms for transactional memory often create conflict sets, which are then scheduled in groups [1, 9]. Hybrid transactional memory (TM) approaches [5, 22, 27] create partitions based on the likelihood of a transaction committing using hardware, rather than software, and extensions to transaction irrevocability [28, 29] will require a similar partitioning based on the behaviors of transactions.

5.1 Algorithm Design

Algorithm 3 presents our Mindicator group mutual exclusion algorithm. There are three data structures. As in our previous algorithms, we maintain a $counter$ and a $mindicator$. However, we now add an $array$ of per-session counters.[2]

The session counter indicates whether a session has any threads that are currently waiting to begin their critical section. This serves two roles. First, a new thread of a particular session can quickly identify the logical queue node to which it belongs (i.e., the integer it should use to arrive in the Mindicator). Second, when a session reaches the position immediately before the head of the queue, it closes the preceding session, so that subsequent threads belonging to that session will order after enqueued operations of other sessions.

In detail, when a session's counter has a value of $\top$, then no threads belonging to that session are enqueued and waiting to execute a critical section. This can either mean that no such operations currently exist, or that the session is at the head of the queue, and operations of that session are executing. In either case, a new operation of that session must begin by creating a new queue node. This is achieved by incrementing $counter$ to generate a new odd value, which also serves to lock the data structure. While the lock is held, the thread arrives with the new odd value and sets its session's counter to that value.

When the session counter sc_i for session i is not $\top$, any thread of session i can join the session if it can atomically set its value in the Mindicator to sc_i before sc_i is reset to $\top$. Clearly, the non-$\top$ counter values are strictly increasing.

The core challenge is when and how to prevent

Algorithm 3: Group Mutual Exclusion with Mindicators

```
shared data
    counter    : Integer
    array      : Integer[]
    mindicator : Mindicator

1  procedure GROUPLOCK(gid : Integer)
2      while true do
3          g ← array[gid]
4          if g ≠ ⊤ then
5              mindicator.ARRIVE(g)
6              g′ ← array[gid]
7              if g = g′ then break
8              mindicator.DEPART()
9          else
10             c ← counter
11             g ← array[gid]
12             if c mod 2 = 0 ∧ g = ⊤ then
13                 if CAS(&counter, c, c + 1) then
14                     mindicator.ARRIVE(c + 1)
15                     g ← c + 1
16                     array[gid] ← c + 1
17                     counter ← c + 2
18                     break
       // wait for my group to be the oldest group
19     while true do
20         curr ← mindicator.QUERY()
21         if curr ≥ g − 2 then
22             if exists i such that array[i] = g − 2 then
23                 array[i] = ⊤
24                 continue
25         if curr = g then break

26 procedure GROUPUNLOCK()
27     mindicator.DEPART()
```

[2] In practice, these counters should be placed on different cache lines to avoid false sharing.

threads from joining sessions that match their session type. The most desirable properties are that (a) a session that is not at the head of the queue can always admit additional threads, and (b) a session that is at the head of the queue can admit additional threads only if it is the only session in the queue. We achieve these properties by closing the session ordered at c in the acquisition phase of a thread of the session at $c + 2$. This ensures that a session never closes prematurely, and also guarantees that session c will be closed before the QUERY that enables threads of session $c + 2$ to begin their critical sections.

5.2 Algorithmic Properties

Hadzilacos [13] establishes four criteria for an ideal group mutual exclusion algorithm, which we consider below:

- Mutual Exclusion: It is clear that two threads can execute their critical sections simultaneously only if they are of the same session. This follows immediately from the manner in which the queue is represented.
- Lockout Freedom: This asserts that every attempt will ultimately be granted, unless a thread remains in its critical section forever. Our algorithm does not provide this guarantee, since a thread may starve in the lock-free Mindicator arrive operation, starve attempting to increment the counter, or block while another thread is opening its session.
- Bounded Exit: This requires a thread to depart from its critical section in a fixed number of instructions. Since we use a lock-free Mindicator operation, bounded exit is not guaranteed. Starvation while departing is possible.
- Concurrent Entering: This places strict requirements on the delay a thread incurs when trying to enter its critical section in the absence of other pending requests from different sessions. The mechanism on Lines 21–24 provides this feature: when there is exactly one session, it remains open indefinitely, but any subsequent session will immediately close an open session that is at the head of the queue.

These limitations draw an interesting contrast worthy of future exploration. The current state-of-the-art [2] implementation achieves these properties through the use of multiple wait-free f-arrays [15] and load-linked/store-conditional (LL/SC) operations that must be simulated via multiple CAS instructions. Additionally, it does so with no more than $\Theta(log(n))$ remote memory references (RMRs), where n is the number of threads. In contrast, we use a single Mindicator, do not require LL/SC, but require in the worst case $O(log(n) + S)$ RMRs, where S is the number of sessions. The former term in this bound comes from the worst-case cost of Mindicator operations. The latter is a consequence of the mechanism for finding and closing a session, but could be eliminated by extending the Mindicator to store tuples instead of integers.

While we mentioned several candidate workloads for group mutual exclusion, no well-known use exists at this time. Should our predictions prove true, then in the common case it is likely that there will be few sessions. To use our hybrid transactional memory example, if the common case is that all threads execute in the same mode, then an important optimization will be to minimize overhead when there is only one session. At the expense of wait-freedom, our algorithm reduces this expected case to code nearly indistinguishable from a readers-writer lock with 100% readers: the counter is never incremented, and thus each acquire consists of two shared memory reads and a Mindicator arrive.

If we use the performance results in Figure 3 as a proxy for this expected case, we can conjecture that our new algorithm will outperform Bhatt and Huang's algorithm: ignoring the additional complexity of LL/SC and multiple f-arrays, it is clear that the Mindicator is more scalable. This is no surprise, since Mindicator operations typically do not modify the root, whereas every f-array operation must.

The degree to which this conjecture generalizes to real-world problems is an open question. On the one hand, wait-freedom is a valuable property, particularly in real-time systems and operating systems. On the other, since most software TM implementations are lock-based, and most hardware TM implementations are obstruction-free at best, a locking but scalable solution to group mutual exclusion may be acceptable for uses within the TM domain.

6 Conclusions

In this paper, we explored the role that Mindicators can play in simplifying and accelerating synchronization algorithms. Our algorithms are simple and scale well, though they offer weaker progress guarantees than previous solutions.

The most promising direction for future work, we believe, is to explore the role that HTM can play in simplifying these algorithms further. In addition to affording lower latency in Mindicators, an HTM-based algorithm could eliminate the need for blocking, thereby eliminating the remaining weakness of our algorithm.

References

[1] Hagit Attiya and Alessia Milani. Transactional Scheduling for Read-Dominated Workloads. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, December 2009.

[2] Vibhor Bhatt and Chien Chung Huang. Group Mutual Exclusion in $O(logn)$ RMR. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing*, Zurich, Switzerland, July 2010.

[3] Guy Blelloch, Perry Cheng, and Phillip Gibbons. Scalable room synchronizations. *Theory of Computer Systems*, 36(5):397–430, 2003.

[4] Pierre-Jacques Courtois, F. Heymans, and David Parnas. Concurrent Control with "Readers" and "Writers". *Communications of the ACM*, 14(10):667–668, 1971.

[5] Luke Dalessandro, Francois Carouge, Sean White, Yossi Lev, Mark Moir, Michael Scott, and Michael Spear. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, March 2011.

[6] Luke Dalessandro, Michael Spear, and Michael L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, January 2010.

[7] Mathieu Desnoyers, Paul McKenney, Alan Stern, Michel Dagenais, and Jonathan Walpole. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, 2012.

[8] Dave Dice, Yossi Lev, Yujie Liu, Victor Luchangco, and Mark Moir. Using Hardware Transactional Memory to Correct and Simplify a Readers-Writer Lock Algorithm. In *Proceedings of the 18th ACM Symposium on Principles and Practice of Parallel Programming*, Shenzhen, China, February 2013.

[9] Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: Scheduling-Based Collision Avoidance and Resolution for Software Transactional Memory. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing*, Toronto, ON, Canada, August 2008.

[10] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. SNZI: Scalable NonZero Indicators. In *Proceedings of the Twenty-Sixth ACM Symposium on Principles of Distributed Computing*, Portland, OR, August 2007.

[11] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, King's College, University of Cambridge, September 2003.

[12] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.

[13] Vassos Hadzilacos. A Note on Group Mutual Exclusion. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing*, Newport, RI, August 2001.

[14] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin Hertzberg. A Scalable Transactional Memory Allocator. In *Proceedings of the International Symposium on Memory Management*, Ottawa, ON, Canada, June 2006.

[15] Prasad Jayanti. f-arrays: Implementation and applications. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, Monterey, California, July 2002.

[16] Prasad Jayanti, Srdjan Petrovic, and King Tan. Fair Group Mutual Exclusion. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, Boston, MA, July 2003.

[17] Patrick Keane and Mark Moir. A simple local-spin group mutual exclusion algorithm. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, Atlanta, GA, May 1999.

[18] Orran Krieger, Michael Stumm, Ronald Unrau, and Jonathan Hanna. A Fair Fast Scalable Reader-Writer Lock. In *Proceedings of the International Conference on Parallel Processing*, Syracuse, NY, August 1993.

[19] Christoph Lameter. Effective Synchronization on Linux/NUMA Systems. In *Proceedings of the May 2005 Gelato Federation Meeting*, San Jose, CA, May 2005.

[20] Leslie Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, 1974.

[21] Yossi Lev, Victor Luchangco, and Marek Olszewsk. Scalable Reader-Writer Locks. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, AB, Canada, August 2009.

[22] Yossi Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased Transactional Memory. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, August 2007.

[23] Yujie Liu and Michael Spear. Brief Announcement: A Nonblocking Set Optimized for Querying the Minimum Value. In *Proceedings of the 30th ACM Symposium on Principles of Distributed Computing*, San Jose, CA, June 2011.

[24] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[25] John Mellor-Crummey and Michael Scott. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. In *Proceedings of the Third ACM Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[26] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1), 1991.

[27] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, June 2011.

[28] Michael Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott. Implementing and Exploiting Inevitability in Software Transactional Memory. In *Proceedings of the 37th International Conference on Parallel Processing*, Portland, OR, September 2008.

[29] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable Transactions and their Applications. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.